Project 1: Transparent Remote File Operations

The project folder has two C files - server.c and mylib.c (client program). The _init() function of the client program connects with the server and returns a socket FD. The server accepts a connections and forks each connection into child process. The parent continues listening for new connections.

I have defined structs in the client and server for structuring and handling the marshalled bytes transferred between client and server. Struct **request_header_t** , is the first header in all the requests made from the client to the server.

```
typedef struct{
   int total_len;
   int opcode;
   unsigned char data[0];
} request_header_t;
```

This contains the value total_len - the entire length of the marshalled data the recv() can expect, opcode – used to identify the operation to be performed by the server and the data[0], which will be used to point the next piece of data in the marshalled buffer. All other structs follow a similar semantics, with the data[0] used as a pointer to the next block in the buffer. This is like a delimiter.

I am also adding an offset of 66666 (macro OFFSET) to the FDs returned by the server to differentiate them from the FDs that are to be handled locally. The system call for FDs with values less than this offset (not received by the server) are handled by using the function orig_XXX. If the value of the FD is greater than or equal to 66666, I subtract 66666 from it and send the value to the server.

The functions are mapped to their respective system calls in the do_process() function in the server.c. **If the return type of a function is a small fixed length variable, I am sending a buffer of size 8 bytes in the response. Where the first 4 bytes is the return value (int, size_t) of the system call and the next 4 bytes is the errno**.

For functions like "**read**", we need to send the buffer with the contents read by the read call. I am passing this buffer with the help of structs pointed by the data[0] pointer (of the read_response_header struct), in a similar fashion as the requests are sent from the client to the server.

For the marshalling and unmarshalling trees I used Depth First Algorithm. Starting from the root I send each node encountered, in a marshalled struct format, to the client. At the client side I use DFS algorithm to receive nodes from the server and construct the tree simultaneously with the receipt of each node. (Recursive call to the recursive_tree() function is done inside a for loop for number of child sub_dirs each directory node expects.)

In case of error when getdirtree() returns NULL, I send a buffer of size 4 to the client with the value of the integer set to 0. This avoids the client and the server from getting into the DFS recursion. I am also using DFS for freeing the tree on the client side. This time starting the free from the leaves and moving up to the root. At server I use the freedirtree() library function.